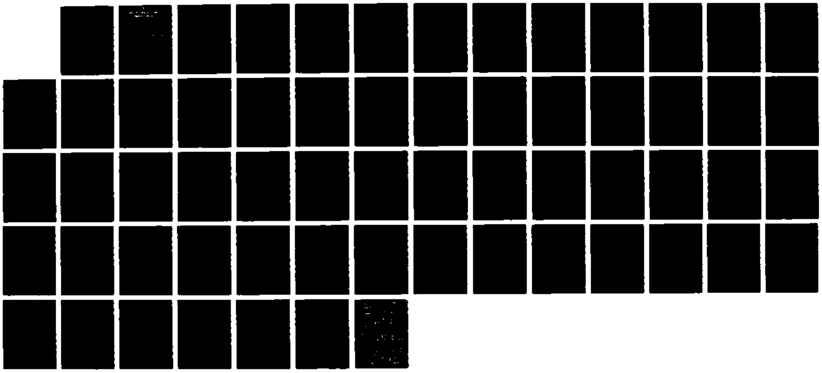
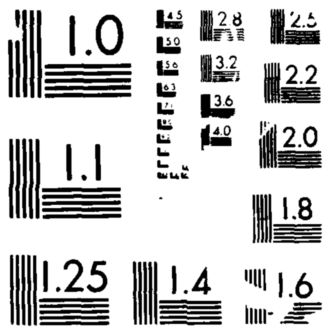
AD-A195 076 A DYNAMIC SCHEDULER FOR A COMPUTER AIDED PROTYPING 1/1
SYSTEM(U) NAVAL POSTGRADUATE SCHOOL MONTEREY CA
S L EATON MAR 88

UNCLASSIFIED F/G 12/5 NL




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3

(2)

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

A DYNAMIC SCHEDULER FOR A COMPUTER AIDED PROTYPING SYSTEM

by

Susan L. Eaton

March 1988

Thesis Advisor Luqi

Unclassified

security classification of this page

REPORT DOCUMENTATION PAGE

1a Report Security Classification Unclassified	1b Restrictive Markings
2a Security Classification Authority	3 Distribution Availability of Report Approved for public release; distribution is unlimited.
2b Declassification Downgrading Schedule	
4 Performing Organization Report Number(s)	5 Monitoring Organization Report Number(s)
6a Name of Performing Organization Naval Postgraduate School / 6b Office Symbol (if applicable) 32	7a Name of Monitoring Organization Naval Postgraduate School
6c Address (city, state, and ZIP code) Monterey, CA 93943-5000	7b Address (city, state, and ZIP code) Monterey, CA 93943-5000
8a Name of Funding Sponsoring Organization / 8b Office Symbol (if applicable)	9 Procurement Instrument Identification Number
8c Address (city, state, and ZIP code)	10 Source of Funding Numbers: Program Element No / Project No / Task No / Work Unit Accession No

11 Title (include security classification) A DYNAMIC SCHEDULER FOR A COMPUTER AIDED PROTOTYPING SYSTEM

12 Personal Author(s) Susan L. Eaton

13a Type of Report	13b Time Covered	14 Date of Report (year, month, day)	15 Page Count
Master's Thesis	From To	March 1988	60

16 Supplementary Notation The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

17 Cosati Codes			18 Subject Terms (continue on reverse if necessary and identify by block number)
Field	Group	Subgroup	rapid prototyping, dynamic scheduling, ada

19 Abstract (continue on reverse if necessary and identify by block number)

Current software development methodologies have proven to be ineffective for meeting the rising demand for fast production of reliable software for hard real-time computer systems. A computer-aided, rapid prototyping system (CAPS) based on a Prototype System Description Language (PSDL) and a set of software tools including an Execution Support System (ESS), has been proposed by other research and provides a promising and cost effective alternative to the traditional development life cycle of these systems.

This study proposes a four function design for the dynamic scheduler of the CAPS ESS. This design includes a method for invoking processes for the ESS static scheduler and translator, a scheduling algorithm for the scheduling of the prototype's non-time critical processes, and a method for error and interrupt handling during prototype execution.

20 Distribution Availability of Abstract ☒ unclassified unlimited ☐ same as report ☐ DTIC users	21 Abstract Security Classification Unclassified
22a Name of Responsible Individual Luqi	22b Telephone (include Area code) (408) 646-2735 / 22c Office Symbol 52LQ

DD FORM 1473,84 MAR 83 APR edition may be used until exhausted. All other editions are obsolete

security classification of this page

Unclassified

Approved for public release; distribution is unlimited.

A Dynamic Scheduler for A Computer Aided Prototyping System

by

Susan L. Eaton
Lieutenant, United States Navy
B.A., Towson State University, 1980

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN TELECOMMUNICATIONS SYSTEMS
MANAGEMENT

from the

NAVAL POSTGRADUATE SCHOOL
March 1988

Author: [illegible]
Susan L. Eaton

Approved by: Luqi
Luqi, Thesis Advisor

Barry A. Frew
Barry A. Frew, Second Reader

[illegible]
David R. Whipple, Chairman,
Department of Administrative Science

[illegible]
James M. Fremgen, Acting,
Dean of Information and Policy Sciences

ABSTRACT

Current software development methodologies have proven to be ineffective for meeting the rising demand for fast production of reliable software for hard real-time computer systems. A computer-aided, rapid prototyping system (CAPS) based on a Prototype System Description Language (PSDL) and a set of software tools including an Execution Support System (ESS), has been proposed by other research and provides a promising and cost effective alternative to the traditional development life cycle of these systems.

This study proposes a four function design for the dynamic scheduler of the CAPS ESS. This design includes a method for invoking processes for the ESS static scheduler and translator, a scheduling algorithm for the scheduling of the prototype's non-time critical processes, and a method for error and interrupt handling during prototype execution.

TABLE OF CONTENTS

I. INTRODUCTION 1
A. BACKGROUND 1
B. OBJECTIVES 7
C. BENEFITS OF THIS STUDY 9
D. OVERVIEW 10

II. BACKGROUND AND DEVELOPMENT OF SCHEDULING ALGORITHMS 11
A. THE SCHEDULING PROBLEM 11
B. SCHEDULING METHODS 12
1. DECOMPOSITION STRATEGIES 12
2. THREE PROCESS MODELS 14
3. EARLIEST DEADLINE-PREDECESSOR PRIORITY ALGORITHM 17
4. THE RATE MONOTONIC SCHEDULING ALGORITHM 17
5. "NEXT-FIT-M" 19
6. A TIME-DRIVEN SCHEDULING MODEL 19
7. DYNAMIC SCHEDULING OF TASK GROUPS 21
8. A RECEIVER-INITITATED SCHEDULING STRATEGY 22
9. APPLICATIONS OF THESE METHODS FOR THE CAPS SCHEDULERS 22

III. THE CAPS DYNAMIC SCHEDULER 23
A. SCHEDULING FUNCTIONS 23
1. THE RUN-TIME EXECUTIVE FUNCTION 23
2. THE CREATE NON-TIME CRITICAL OPERATOR SCHEDULE FUNCTION 25
3. THE TERMINATE PROTOTYPE FUNCTION 28
4. THE HANDLE INTERRUPTS FUNCTION 28
B. THE USE OF ADA 29

IV. SUMMARY 31

A. THE QUESTIONS ANSWERED 31
B. THE PROBLEMS THAT REMAIN 31
C. CAPS: AN EFFECTIVE DEVELOPMENT ALTERNATIVE 32
D. CONCLUSION 33

APPENDIX A. A PSDL PROTOTYPE 34

APPENDIX B. PSDL GRAMMAR SUMMARY 38

APPENDIX C. DYNAMIC SCHEDULER FUNCTIONS 41

APPENDIX D. PSEUDO-CODE FOR AN ADA PROGRAM 43

LIST OF REFERENCES 49

INITIAL DISTRIBUTION LIST 52

LIST OF FIGURES

Figure 1. PROCESS OF REQUIREMENTS DETERMINATION 3
Figure 2. CAPS Architecture 6
Figure 3. COMPONENTS OF THE CAPS EXECUTION SUPPORT SYSTEM . 8
Figure 4. DYNAMIC SCHEDULER FUNCTIONS 24
Figure 5. STATIC SCHEDULER BLOCKING METHOD 26
Figure 6. CREATION AND EXECUTION OF THE DYNAMIC SCHEDULE 27

I. INTRODUCTION

A. BACKGROUND

Increasing demand for rapid development of high quality software has risen to the point that significant improvements must be made to current software development methodologies. This is because these methods do not produce software fast enough, nor do they result in software products of sufficient quality. This is particularly true for development of software for hard real-time systems. A hard real-time system is one in which tasks have deadlines that must be met, otherwise severe consequences may result. Many Command, Control and Communications (C3) Systems are examples of such systems.

Production of hard real-time systems that support communications requirements within the area of C3 are particularly challenging to software developers. One reason for this is that communications systems are usually subject to very stringent real-time requirements. For example, receiving and processing data from remote sensors may need to occur in the micro or millisecond timeframe. Another reason, often inherent to defense systems, is that communications software (as well as other types of software) must be interoperable across a wide variety of hardware and software environments. This is exemplified by the fact that equipment from multiple vendors (utilizing proprietary or incompatible protocols), and obsolete, poorly documented systems must function together in support of various operational requirements. Furthermore, maintenance considerations across these diverse environments introduce an additional level of difficulty for software developers because the interoperability of these systems must be maintained when inconsistencies are reconciled or when upgrades are applied.

One method for meeting these challenges, and the increased demand for rapid system development, is rapid prototyping. A prototype is an executable model or pilot version of the intended system which is used as an aid in analysis and design rather than as production software to be delivered to the user. Rapid prototyping is the construction activity which creates this executable model. This technique has been found to be effective for clarifying user requirements and eliminating the large amount of wasted effort currently spent on developing software to meet incorrect or inappropriate requirements in traditional software life cycles. [Ref. 1: p. 1]

Rapid construction of executable prototypes for hard real-time systems would be greatly enhanced through the use of a computer-aided design system. One such system proposed by [Ref. 2] and [Ref. 3] is the Computer Aided Prototyping System (CAPS). CAPS presents an alternative to the traditional software development life-cycle and is based on a Prototype System Description Language (PSDL) and a prototyping methodology.

The CAPS prototyping methodology, as illustrated by Figure 1 on page 3 is an iterative process. The software developer constructs a prototype based on user requirements, then the developer and user examine the executable prototype together. During this examination, adjustments are made and the prototype is modified until both the user and developer agree that the user's requirements will be met.

Prototype System Description Language (PSDL) was developed in conjunction with this methodology because a language for supporting rapid prototyping of large real-time systems has different requirements from general purpose programming or specification languages. PSDL contains several unique features which meet these requirements. For example:

PSDL is based on a simple computational model which limits and exposes the interaction between system modules thus promoting effective modularization of the prototype.

PSDL contains basic data, control, and function abstractions which allow specification and representation of the intended system most important for creation and execution of the prototype.

Appendix A is an example of a PSDL prototype as it appears in [Ref. 4: pp. 27-40] and Appendix B is a summary of PSDL grammar and language conventions from [Ref 1: pp. 54-56], provided as additional clarification for this example. This prototype was developed to model a simple system for treating brain tumors using hyperthermia and was structured to meet the following requirements:

1. Shutdown: Microwave power must drop to zero within 300 milliseconds of turning off the treatment switch.
2. Temperature Tolerance: After the system stabilizes, the temperature must be kept between 42.4 degrees C. and 42.6 degrees C.
3. Maximum Temperature: The temperature must never exceed 42.6 degrees C.
4. Startup Time: The system must stabilize within 5 minutes of turning on the treatment switch.
5. Treatment Time: The system must shut down automatically when the temperature has been above 42.4 degrees C. for 45 minutes.

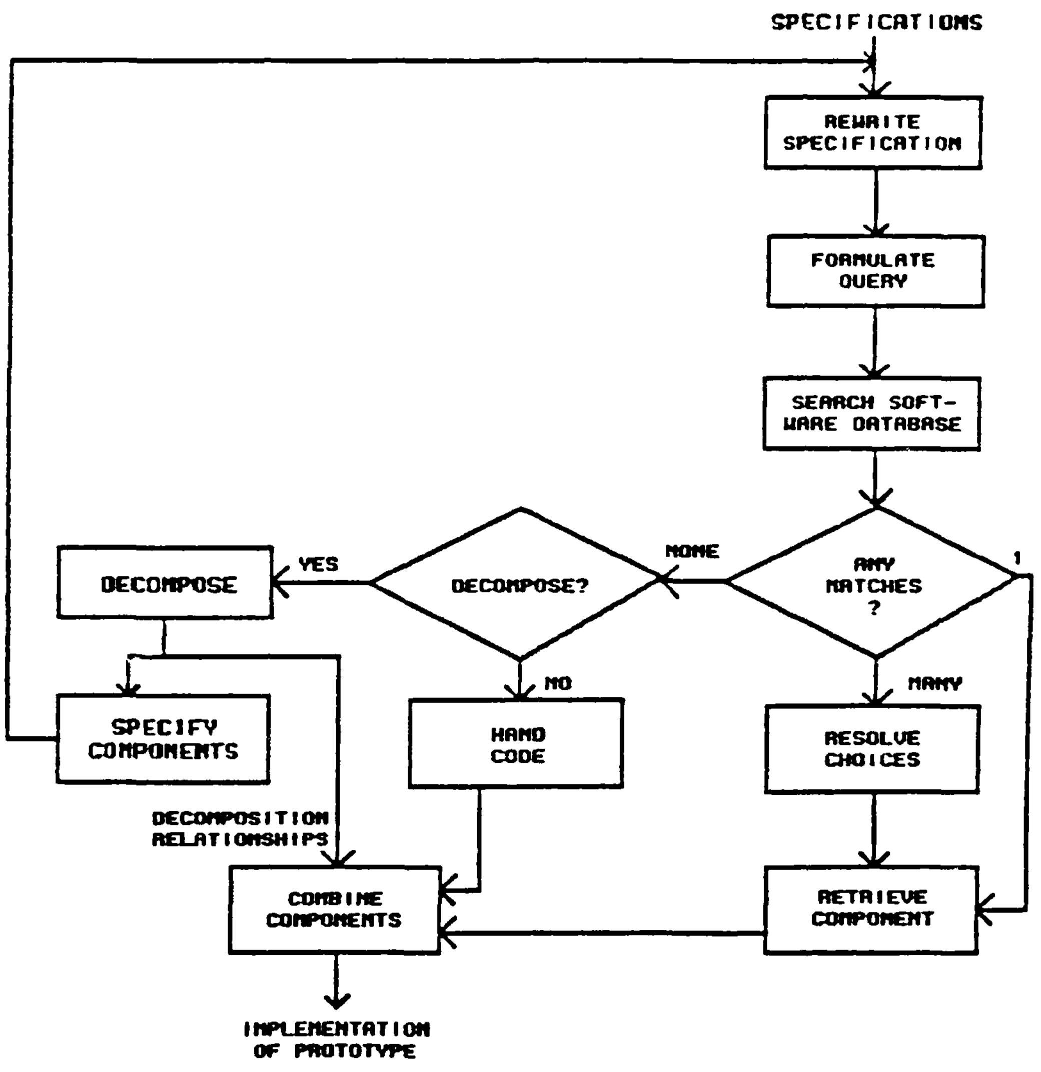


Figure 1. PROCESS OF REQUIREMENTS DETERMINATION AND VALIDATION BY PROTOTYPING

[Ref. 4: pp. 26-27]

A prototype is created in PSDL using networks of operators communicating via data streams. A data stream is a communications link connecting exactly two operators, a producer (one which produces a data value), and a consumer (one which consumes or receives the data value). Data streams also carry data values which represent EXCEPTION conditions. PSDL exceptions are values of a built in abstract data type called EXCEPTION. This type has operations for creating an exception with a given name (e.g. "overflow"), and for detecting whether a value is normal (i.e. belongs to some data type other than EXCEPTION). [Ref. 4: p. 13]

The other PSDL data types include the unalterable subset of the built-in types of the Ada® programming language (Ada® is a registered trademark of the United States Government, Ada Joint Programming Office), user defined abstract types, the special type TIME (the other special type being EXCEPTION as previously described), and the types that can be built using the immutable type constructors of PSDL. The PSDL type constructors were chosen to provide powerful data modeling facilities with a small set of semantically independent structures. [Ref. 4: p. 15]

Each data type or operator is either composite or atomic. Composite operators are implemented by decomposing them into networks of more primitive operators (using PSDL). Atomic operators are created by retrieving an implementation from a software base containing reusable software components implemented in an underlying programming language.

In in order to meet timing constraints of the prototype under construction, an operator can either be periodic, or sporadic. A PSDL operator is periodic if a period has been specified for it explicitly, or if it inherits a period from a higher level in the decomposition of the hierarchical prototype. If neither of these conditions are true, then the operator is sporadic or data driven. A sporadic operator is executed (triggered by) the arrival of a new data value, possibly at irregular time intervals, whereas periodic operators are triggered or executed at regular time intervals (specified periods). A periodic operator must be completed sometime between the beginning of the period and a deadline (which defaults to the end of the period). Periodic operators have traditionally been the basis for the design of most real-time systems, but the importance of data driven operators for this type of system is also beginning to be recognized since event *driven in terms of informal software design methodology, or interrupt driven in terms* of hardware language, can be treated in this category. [Ref. 4: pp. 6-13]

The foregoing features make PSDL particularly appropriate for real-time system design. Its structure is highly suitable for multiple modifications during prototyping iterations because it consists of basic building blocks that allow descriptions of abstractions through a top-down design based on data flow. Additionally, the formal structure of PSDL for specifying the user's real-time constraints provides a basis for automating the production code to an underlying programming language e.g. Ada®. The execution of the PSDL prototype also verifies that the design of an embedded system (a system that is part of a larger system such as a guidance computer on a missile), within given timing constraints for the prototype components, will interact with its environment in a way that meets the timing constraints of the entire system. [Ref. 1: p. 3]

The other components of the CAPS are user interfaces, including a syntax directed editor with graphics capability (for speeding up design entry and preventing syntax errors), an execution support system for demonstrating and measuring prototype behavior and for performing static analyses of the prototype design, a software design management system for retrieving and adapting reusable software components, and a component base which functions as a repository for the reusable components [Ref. 2: p. 9]. The reusable software components in the software base can be written in any general purpose programming language (provided that PSDL specifications for each module are included). Figure 2 on page 6 illustrates the CAPS architecture.

For purposes of simplification, and because of its required use within the Department of Defense as a standard development language, Ada® has been chosen for implementing both the reusable components in the software base and the PSDL execution support environment. Ada® is a powerful programming language that provides unique features not found in other languages. These include exception handling, inter-task communication, (both of which will be demonstrated to be particularly important to the CAPS execution support environment), and facilities such as generic packages (reusable software components). Several predefined generic units are already included as part of the Ada® language definition e.g. CALENDAR which can be used to provide date and time information. [Ref. 5: pp. 33-34]

An Ada® program is composed of one or more program units, most of which may be separately compiled. Program units consist of subprograms, tasks, packages, and generic units. A subprogram is either a procedure or a function. A procedure specifies a sequence of actions and is invoked by a procedure call statement. A function specifies a sequence of actions and also returns a value called the result; therefore a function call is an expression. A task, on the other hand, defines an action that is logically executed in parallel with other tasks. A task may be implemented on a single processor, a multiprocessor, or a network of computers. A package is a collection of computational re-

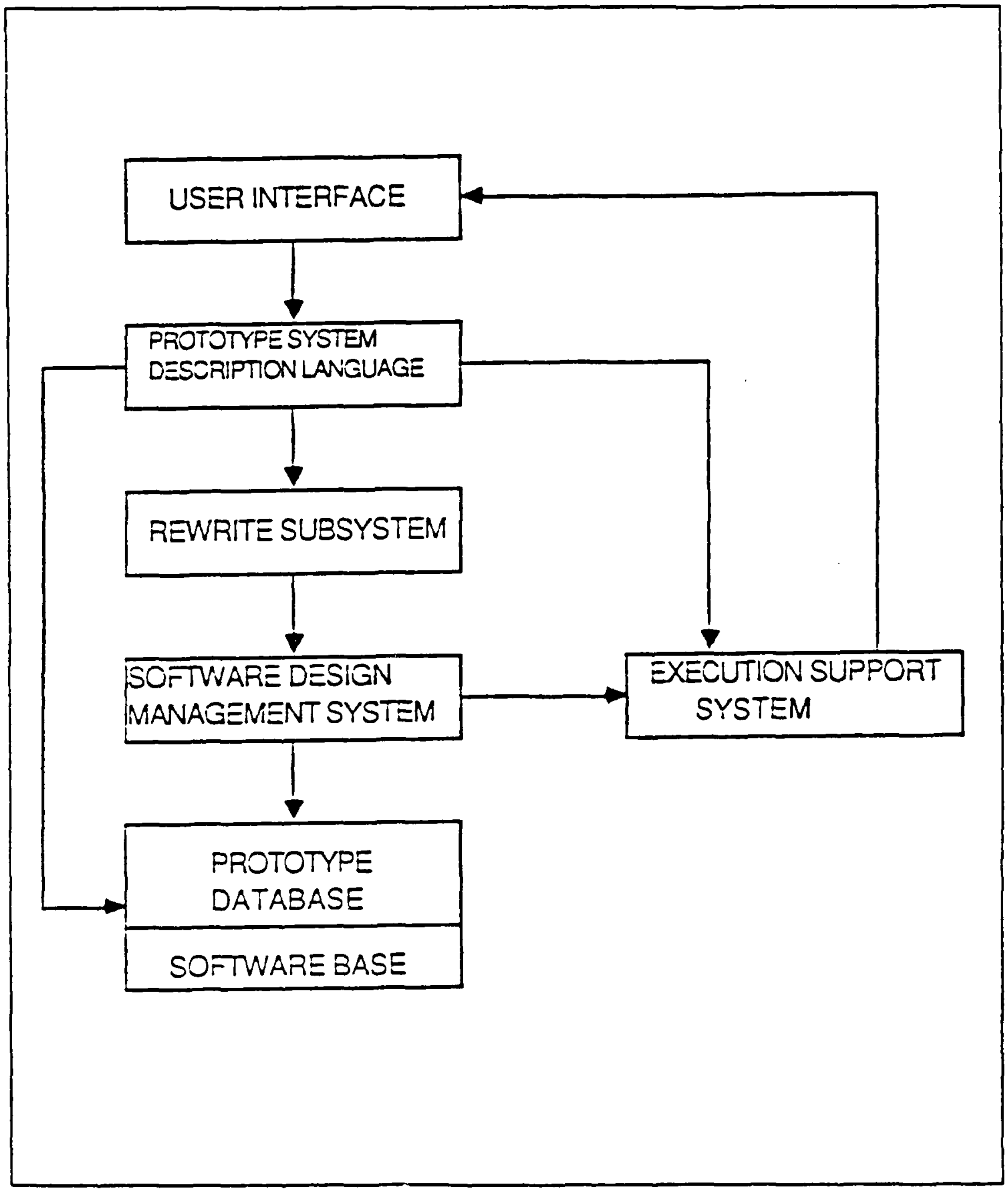


Figure 2. CAPS Architecture

sources, which may encapsulate data types, data objects, subprograms, tasks, or even other packages. Its primary purpose is to express and enforce a user's logical abstractions within the language. A generic unit is a "template" or "pattern" for subprograms and packages and serves as the primary mechanism for building reusable software components. Use of a generic unit within an Ada® program is termed instantiation.

All Ada® program units generally have a similar two-part structure, consisting of a specification and a body. The specification identifies the information visible to the client (interface) of that program unit and the body contains the unit implementation details. [Ref. 5: pp. 55, 554]

Ironically, it is some of these same attractive features of the language that make Ada® too complex and hence, too impractical, for its direct use in the rapid prototyping environment. PSDL however has incorporated many of the desirable features of Ada® while eliminating the associated complexity. The abstractions of PSDL allow a system designer to express ideas at the specification and design level rather than at the programming language level. This substantially reduces the need for consideration of lower-level details and flow control that would be required if the prototype was developed using Ada® directly.

B. OBJECTIVES

The primary focus of this study is the conceptual development of one component of the execution support system of the CAPS, the dynamic scheduler. As it is currently proposed, the execution support system will be comprised of three components, a translator, a static scheduler, and a dynamic scheduler. The translator is developed in [Ref. 6] and the static scheduler is developed in [Ref. 7] and [Ref. 8]. A secondary, but equally important focus is the interfacing of the dynamic scheduler with these other two components.

Within the CAPS execution support environment each of these components will perform several functions as shown in Figure 3 on page 8. The translator has four main purposes:

1. To augment the PSDL code
2. To implement PSDL data streams
3. To implement PSDL conditionals (triggering conditions)
4. To implement PSDL timers (accomplished through the use of a standard library package which communicates with a hardware clock and is included in any prototype that uses timers)

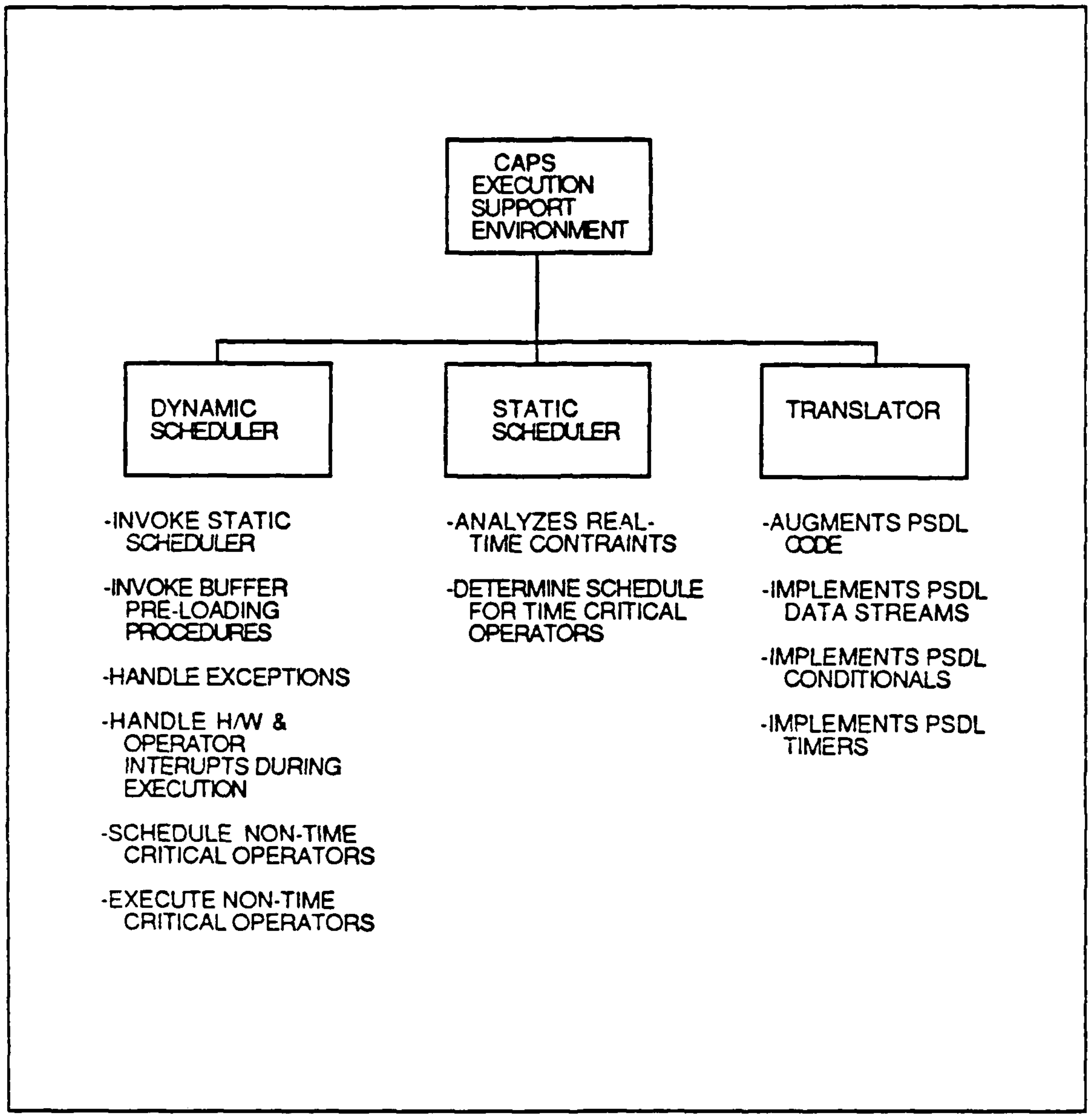

CAPS EXECUTION SUPPORT ENVIRONMENT
DYNAMIC SCHEDULER
STATIC SCHEDULER
TRANSLATOR
-INVOKE STATIC SCHEDULER
-INVOKE BUFFER PRE-LOADING PROCEDURES
-HANDLE EXCEPTIONS
-HANDLE H/W & OPERATOR INTERUPTS DURING EXECUTION
-SCHEDULE NON-TIME CRITICAL OPERATORS
-EXECUTE NON-TIME CRITICAL OPERATORS
-ANALYZES REAL-TIME CONTRAINTS
-DETERMINE SCHEDULE FOR TIME CRITICAL OPERATORS
-AUGMENTS PSDL CODE
-IMPLEMENTS PSDL DATA STREAMS
-IMPLEMENTS PSDL CONDITIONALS
-IMPLEMENTS PSDL TIMERS

Figure 3. COMPONENTS OF THE CAPS EXECUTION SUPPORT SYSTEM

The static scheduler analyzes the real-time constraints declared in the PSDL prototype and attempts to find a static schedule meeting the constraints of the time critical operators of the prototype under construction.

The dynamic scheduler performs four major functions for the CAPS execution support system. The first function, which is to act as a "run-time executive", is of particular importance to the other two CAPS components. As the run-time executive, the dynamic scheduler will invoke the static scheduler, and it will invoke buffer pre-loading procedures required by the translator for implementation of data streams. Two other functions include exception handling and hardware or operator interrupt handling that may occur during prototype execution.

The fourth and perhaps most important function of the dynamic scheduler will be the scheduling and execution of the PSDL operators which are not time critical (i.e. do not have real-time constraints). This schedule will be constructed and executed during prototype execution using "spare processing time" created as a result of early completion of time critical operators by the static scheduler. Because PSDL assumes that time constraints of critical operators are absolute when given, the static scheduler allocates processing resources based on worst case or maximum execution times. On the average, these worst case processor loads tend to be rare. When a time-critical operator or group of operators finishes executing before this worst-case time allocation, the static scheduler can "transfer" control of processor resources to the dynamic scheduler in order to utilize the resulting spare capacity.

The requirement for explicitly passing control to the dynamic scheduler when the static scheduler reaches an idle state is necessary because the Ada® language does not have features for determining when a task or process with an undefined priority should be executed [Ref. 5: p. 282]. Once control of processing resources is passed to the dynamic scheduler, spare processing capacity can be allocated among the non-time critical operators based on a scheduling process that is not restricted by the requirement for meeting real-time constraints.

C. BENEFITS OF THIS STUDY

The benefits to be derived from this study are twofold. The first of these is that development of a dynamic scheduler for the proposed CAPS aids in meeting the need for development of a rapid prototyping tool. An effective CAPS would result in significant improvements and cost savings in the development of hard real-time software systems which support C3 mission requirements as well as software development for other DOD, and private industry applications.

The second benefit is the focus placed on more effective processor utilization as a result of scheduling non-time critical tasks or processes during slack or spare processing periods. Previous research in the area of real-time system scheduling has greatly emphasized, and rightly so, the requirement for meeting the real-time constraints of a system or network of systems. This particular emphasis has minimized the importance of processor under-utilization which often occurs as a result of ensuring that real-time constraints are met. The problem of under-utilization is wasteful and could become quite costly if it is allowed to occur on a regular basis. Design and interface of a dynamic scheduler for use within the rapid prototyping environment may provide a viable solution to this problem.

D. OVERVIEW

The remainder of this study is described by the following overview:

A survey of the background and development of scheduling problems and algorithms

Development of a dynamic scheduler based on concepts provided by this survey and the use of Ada® as an implementation language

A summary which describes the questions answered by this study, future questions or design areas that need to be addressed, and a brief description of a communications system for demonstrating the feasilibity of the CAPS as a computer-aided design tool.

II. BACKGROUND AND DEVELOPMENT OF SCHEDULING ALGORITHMS

A. THE SCHEDULING PROBLEM

A scheduling algorithm provides a set of rules that determine a process or group of processes to be executed at a particular point in time on a process control computer system or for a network of systems [Ref. 9: p. 194]. Criteria which have historically been used to generate process schedules include maximizing process flow (i.e. minimizing the elapsed time for the entire processing sequence), or minimizing the maximum lateness (lateness is defined to be the difference between the time a process is completed and its deadline when the deadline is missed) [Ref. 10: p. 112].

Development of an algorithm which focuses on maximizing process flow is applicable to the problem of scheduling PSDL operators without real-time constraints since optimal use of idle processing time is an objective of the CAPS dynamic scheduler. However, minimizing lateness is not a consideration for the dynamic scheduler since operators which are not time critical don't have deadlines to meet. For meeting the requirements of the CAPS static scheduler, neither of these criteria is important particularly since operators with real-time constraints are by definition not allowed to be late. The criteria which are important for process scheduling within the CAPS execution support environment include meeting the deadlines of operators with real-time constraints, ensuring that no data loss occurs, and making optimal use of spare processing resources. Clearly, finding or developing scheduling algorithms which optimize this set of criteria presents an interesting and difficult problem.

Another previously defined [Ref. 9: pp. 194-199] consideration for generating process schedules and developing scheduling algorithms is based on precedence or priority of processes to be executed. Two primary priority classifications are static priority and dynamic priority. In the first case, priorities and start times of processes are known in advance and is not expected to change during execution [Ref 9: p. 194]. Within the CAPS, a scheduling algorithm based on a static priority scheme will be used by the static scheduler to create a schedule that meets the timing and precedence relationship requirements for the time critical operators. In the second case, priorities of processes change from time to time, depending upon certain execution conditions (e.g. the availability of idle processing capability) [Ref 9: p. 194]. This priority scheme will be used by the CAPS dynamic scheduler to schedule non-time critical PSDL operators and to perform other functions during prototype execution such as exception or interrupt handling.

B. SCHEDULING METHODS

The requirement for different types of schedulers and scheduling algorithms has been examined in a myriad of research. Most of this work has been directed at the problem of scheduling processes or operations which must meet critical or real-time deadlines, but these efforts also have relevance to the problem of scheduling processes which don't have real-time constraints. The primary reason for this is that while an individual process (e.g. a PSDL operator) may not have a time critical deadline, scheduling of the process or group of processes should be completed within a predetermined block of idle processing time in order to make optimal use of this spare capacity. The following examination and description of scheduling research provides a basis for designing a dynamic scheduler to meet this objective.

1. DECOMPOSITION STRATEGIES

A primary consideration in solving the scheduling problem is how to decompose a set of operations (computations) into a schedule which meets the real-time constraints of a given system or program. Mok in [Ref. 11 : pp. 125-133] proposes three strategies for the decomposition of a set of computations based on timing constraint specifications. Each of these strategies uses a "graph" model to describe the set of computations and a "process" model to describe the output generated by the translation of the set of computations.

The graph model consists of a communications graph, a task graph, and a set of timing constraints. Timing constraints are represented by the expression (t, t + d) where t is the start time for a process, d is its deadline, and t + d the interval or period in which the process is executed. A task graph defines the precedence relationship among computational events that must occur in order to satisfy a given timing constraint. It is composed of "nodes" and "edges" which respectively denote corresponding functional elements and transmission paths for data in the communications graph [Ref. 11: p. 126]. The objective of this structure is to ensure that data flow requirements are met. This is also one of the objectives of the PSDL structure, (the other objective being that real-time constraints will be met). PSDL is based on these concepts with an operator representing a "functional element" of the language, and with data streams representing communications paths which transmit or exchange information between operators.

The Process Model is generated by the translation of the time-critical computation requirements of a real-time system. The result of the translation is a set of time-critical concurrent processes [Ref. 11: p. 125]. The translation that results in the process model is analogous to the generation of the of the CAPS static schedule since this schedule provides the means for meeting a system's real-time constraints.

Based on these concepts, the first strategy to be discussed is Decomposition by Critical Timing Constraints. This strategy works in the following manner. For a particular program, periodic and sporadic processes are created to meet given timing constraints. The period and deadline attributes of a process are set to the corresponding parameters of the timing constraint (t, t + d). These processes may have functional elements in common so a monitor is created to ensure mutual exclusion on the execution of any program element called by two or more processes. When a program created in this manner is executed, each process is executed according to its specified timing constraints even though this may result in duplicate execution of certain computational events. [Ref. 11: p.128]

This strategy works fairly well on single processor with any scheduling discipline as long as the processor doesn't idle while there is an activated process [Ref. 11: p. 128]. The disadvantages associated with the use of this strategy are the duplication of some computations within processes that have compatible timing constraints and the communications costs involved for enforcing mutual exclusion.

A second strategy, Decomposition by Centralized Concurrency Control works in the following way. Periodic timing constraints that are compatible with one another are grouped together. Two periodic timing constraints are compatible if their deadlines (d) are equal, (e.g. d1 = d2), if their task graphs have some nodes in common, and if the period (p) of one can divide, or be divided by the period of the other (p1/p2 or p2/p1). The compatibility relation partitions the periodic timing constraints into a set of equivalence classes. For each equivalence class, a periodic process of compatible periodic timing constraints is created, and a sporadic process is created for each asynchronous timing constraint.

In general, this strategy improves efficiency two ways. First, by merging the computation of compatible timing constraints into a single process, redundant computation can be eliminated. Second, since concurrency control is being centralized, processes tend to be independent of one another and the interprocess communication overhead required for concurrency control will be smaller. One disadvantage associated with this strategy is that attempts to merge compatible timing constraints into a single program by eliminating as much redundant computation as possible, may not yield the shortest program possible. A second disadvantage associated with this strategy is its complexity, which makes it more difficult to understand and to modify when changes are required. [Ref. 11: pp. 129-130]

The third strategy is Decomposition by Distributing Concurrency Control. In this strategy, a periodic process will be created for each node (functional element) in the communication graph. Since a functional element F, may occur in two or more task graphs, the periodic process created for F will be assigned a period attribute equal to the smallest period among the periodic timing constraints in which F occurs. When a periodic process PF, is activated, it first synchronizes with an appropriate set of processes preceded by it. A sporadic process is created for each asynchronous timing constraint as before. If a functional element occurs in both a periodic timing constraint and an asynchronous timing constraint, then a monitor is created to enforce mutual exclusion on the execution of the corresponding program element. [Ref. 11: p. 131]

Use of this strategy results in the following advantages. By assigning a separate process to each functional element, an attempt is made to maximize the computation that can be performed in parallel. Redundant computation is reduced since task graphs of compatible timing constraints that contain the same functional elements are detected in the construction of the synchronization code for each periodic process. If as many processors are available as there are processes, then this strategy can accommodate a wider range of timing constraints than the other two strategies. The primary disadvantage with this approach is again one of complexity and the resultant modification difficulties its use implies. [Ref. 11: p. 132]

2. THREE PROCESS MODELS

Another study by Mok [Ref. 12: pp. 5-17] develops three process models using various scheduling algorithms and techniques. These models are based on the idea that there is a need for an off-line scheduler and a run-time scheduler for meeting the periodic and sporadic timing constraints of most real-time systems. As defined by this work, the off-line scheduler examines the instance of a process, or system and creates a run-time scheduler together with a database for making scheduling decisions at run time. The run-time scheduler is the code for allocating resources in response to requests generated at run time, e.g. timer or external device interrupts. A run-time scheduler is totally on-line if its decisions do not depend on prior knowledge of the future request-times of the processses. A run-time scheduler can also be clairvoyant, which means that it can predict with absolute certainty, the future request times of all processes. A clairvoyant scheduler represents the best possible case though and is usually impossible to implement in practice. And finally, a run-time scheduler is optimal if it always produces a feasible schedule whenever it is possible for a clairvoyant scheduler to do so.

The first model described by this piece of research is the Independent Process Model. It was shown that two possible algorithms provided effective scheduling techniques for this model, the earliest deadline algorithm and least slack algorithm. The earliest deadline algorithm runs any ready process with the nearest deadline and the least slack algorithm runs any ready process which has the least slack time available before it will miss its current deadline. In both cases, ties are broken arbitrarily and the assumption is made that the scheduler can choose to preempt a process by any other ready process at integral time instants.

Although both of these algorithms are effective, the preceding assumption illustrates why neither of them represents an optimal scheduling method. In order for these techniques to be optimal, the scheduler would have to be clairvoyant. For example, the position of an aircraft is updated by a periodic process which computes the X and Y coordinates from sensor measurements. A sporadic process may read the X value, be preempted by the tracking process, and then read a new Y value which is inconsistent with the original X value. Clairvoyancy implies that an exact prediction could be made as to when the sporadic process which updates the X value will occur, which is unlikely. A possible means for eliminating this inconsistency is to prevent processes from preempting one another, but enforcement of such a mutual exclusion constraint results in significant decreases in processing efficiency. [Ref. 12: p. 7]

A feasible, yet still not optimal, alternative to this approach is provided by the Deterministic Rendezvous Model. This model attempts to alleviate the problems associated with the Independent Process Model by using the earliest deadline algorithm with dynamically assigned (determined during execution) process deadlines, and through the implementation of an Ada®-like rendezvous primitive (communications instruction).

The rendezvous primitive establishes synchronization and precedence relationships among executing processes. It operates on the same principle that is required for the establishment of certain data communications links. For example, if Process A wishes to communicate or rendezvous with Process B, A executes a rendezvous primitive. A must then wait for B to execute a rendezvous which indicates that it is ready to exchange information or rendezvous with A. The precedence relationships among processes are created by the requirement that all the computation before the rendezvous primitive in each process must precede all the computation after the corresponding rendezvous primitive in the other process [Ref. 12: p. 9].

At run-time, this model works in the following way. Processes are grouped into scheduling blocks with each block initialized with a deadline. During execution, the deadline of a scheduling block can be moved up if the block must precede another block which has a nearer deadline but which is not yet ready to run. The rendezvous primitive provides the required synchronization and precedence information which allows this scheme to work. It should be pointed out though that this primitive does not guarantee mutual exclusion for a scheduling block. It also cannot be used to establish communications between a periodic process and a sporadic process since by definition, a periodic process must be executed regularly while a sporadic process may never be executed. [Ref. 12: pp. 9-10]

The third model differs only slightly from the Deterministic Rendezvous approach. This model called the Kernelized Monitor, uses an operating system kernel as a monitor for enforcing mutual exclusion of processes during execution. Processor time is allocated only in uninterruptible quantums, say of size q, with q chosen to be bigger than the largest monitor. For simplicity, the required computation times for process scheduling is in exact multiples of q so that each process takes an integral number of quantums to execute. A process to be executed forms a chain of mini-scheduling blocks each of which requires a quantum (the basic time unit of processor allocation). These mini-scheduling blocks form a partial order imposed by the (intra and interprocess) precedence relationships and each is given a request-time and deadline. The mini-scheduling blocks are executed using the earliest dynamic deadline algorithm as previously described in the discussion of the Deterministic Rendezvous Model.

One difference between the execution of mini-scheduling blocks and the execution of blocks created by the Deterministic Rendezvous approach is that preemption should only be allowed to occur after a mini-block has been allocated an integral number of time quantums. Another difference is that between each chain of mini-scheduling blocks an interval called a "forbidden region" is included in in the schedule. The purpose of this interval is to create idle processing time during which a scheduler should not allocate a new quantum of processor time to any process so that a future deadline can be met. [Ref. 12: pp. 10-11]

3. EARLIEST DEADLINE-PREDECESSOR PRIORITY ALGORITHM

Another research effort by Mok demonstrates the use of the earliest deadline algorithm in a slightly different way. This effort was directed at periodic real-time systems where input data arrives at fixed rates, but otherwise there are no explicit timing constraints. Its application is also limited to uniprocessor environments.

The Earliest Deadline-Predecessor Priority (ED-PP) scheduling procedure can be described by these steps. First, a very simple method (as compared to use of the rendezvous primitive) is used to determine precedence relationships among processes. Specifically, processes are ranked in a topological order of their corresponding functional elements in a graph model such that whenever two processes have the same deadline, higher priority is given to the process which appears earlier in the topological ordering (hence the name predecessor priority) [Ref. 13: p. 184]. Next, a round robin scheduler is employed in the following way. Assume that a quantum (the previously defined time unit) is composed of infinitely many slices. A round robin scheduler allocates c/p slices of each quantum to each process P. Each P will be guaranteed to receive c quantums of processor time in every period of length p, thus meeting its deadline. The above allocation can always be done because available processor time U is < = 1. [Ref. 13: p. 186]

The round robin schedule is then transformed into the desired schedule by swapping time slices in the following manner. At any quantum, let P be the process with the nearest deadline as chosen by the ED-PP scheduler. Then, swap as many slices of P from the next quantums as needed to fill just the quantum under consideration. No process will miss its deadline since the deadline of p is the nearest. This swapping is repeated one quantum at a time from the beginning of the schedule until the valid ED-PP schedule of desired length is obtained. [Ref. 13: p. 186]

4. THE RATE MONOTONIC SCHEDULING ALGORITHM

This algorithm works in the following way. For a set of periodic tasks, a fixed priority is assigned to each task, with a higher priority being assigned to tasks with shorter periods. The rate monotonic algorithm is an optimal static priority algorithm in a uniprocessor environment with a set of n tasks with total utilization less than or equal to $n(2^{1/n} - 1)$. When n becomes large, this bound approaches ln 2 (approximately 70%).

One method for implementing this algorithm incorporates a "time-division multiplexing" scheme to schedule periodic tasks (processes). This approach is similar to the round robin scheduler used by the ED-PP algorithm. This is accomplished through the creation of a set of time division multiplex (TDM) slots and then "hand-packing" all the important tasks into them. This is typically done in the context of a cyclical executive (the cyclical executive operates like the round robin scheduler), which generally uses few frequencies. The fastest cycle is usually called the major cycle and the slower ones are called minor cycles. The major cycle is assigned the highest priority. Given the highest priority, a major cycle with period P will be regularly given 1 slot every P units of time. This in effect creates a virtual processor with processing bandwidth 1/P. The period of the major cycle is determined by two factors. First period P must be short enough so that it can accommodate the highest frequency periodic tasks. Second, the major cycle must also accommodate tasks which have lower frequencies but are critical to the mission at hand, since the major cycle has the highest priority. A handcrafted table is then constructed to schedule both the high frequency tasks and the critical tasks over the virtual processor. The construction of the scheduling table often takes many iterations, over the adjustment of the period of the major cycle, the modification of the scheduling table and the optimization of the code of certain tasks. [Ref. 14: pp. 184-185]

Using another approach, this algorithm can be employed to schedule aperiodic (sporadic) tasks. Aperiodic tasks consist of a stream of jobs arriving at the processor according to some random process such as the Poisson pr cess. In this case, there is no deterministic upper bound on the worst case processor utilization task even though each job of an aperiodic task has a bounded worst case execution time. Thus, it is impossible to guarantee that every job's deadline in an aperiodic task will be met. The concept behind dealing with aperiodic tasks is to reserve adequate processor time for each group of tasks so that fast average response time can be ensured.

A simple way to realize this objective is to create a set of periodic tasks, each of which serves a group of aperiodic tasks. Each of these server periodic tasks will be run according to the basic principle of the rate monotonic algorithm. Associated with each server periodic task, there is a ready queue for associated aperiodic jobs. Each of these aperiodic jobs in the associated ready queue will be treated as if it is a periodic job of the server periodic task and dispatched accordingly. That is , if a periodic server has period P and nominal computation time C, then the associated aperiodic job can be executed C time units in every period P at the priority level associated with period P, The ration C/P represents the processor time allocated for associated aperiodic tasks. [Ref. 14: p. 183]

5. "NEXT-FIT-M"

NEXT-FIT-M is better classified as a decomposition strategy than a scheduling algorithm. It was developed for use in conjunction with the rate-monotonic algorithm in a multi-processing environment. The requirement for this strategy is based on the fact that the rate-monotonic algorithm behaves poorly in multiprocessor systems if the rule is followed of not allowing a processor to idle when there is a task ready for execution.

NEXT-FIT-M is based on the following assumptions:

1. Tasks are time-critical and the requests of each task are periodic, with a constant interval between tasks.
2. Deadlines consist of runability constraints only, i.e. each request must be completed before the next request of the of the same task occurs.
3. The tasks are independent in that the requests of a task do not depend on the initiation or the completion of the requests of other tasks.
4. Computation time for the requests of a task is constant for the task. Computation time here refers to the time a processor takes to execute the request without interruption.
5. Task utilization is defined by two numbers, the computation time of the request(c), and the request period(t). The ratio c/t is called the utilization factor of the task. [Ref. 9: p.194]

In a multiprocessor environment, this utilization factor provides a means for decomposing tasks into classes. A class is defined for each available processor in the system, and tasks belonging to a given class are scheduled on the processor with the appropriate class designation. Task classes are created based on a range of utilization factors e.g. class A tasks have utilization factors between .4 and .1, class B tasks range between .2 and .4, etc.. Actual utilization ranges are established using a logarithmic scale derived from the formula $n(2^{1/n} - 1)$ as described by [Ref. 9: p. 195]. When decomposition and assignment of task classes to processors is complete, execution proceeds on each processor according to the rate-monotonic algorithm.

6. A TIME-DRIVEN SCHEDULING MODEL

Another approach to scheduling is illustrated by the Time Driven Scheduling Model and its two associated algorithms, BEValue1 and BEValue2. [Ref. 10: pp. 112-122] This model is based on a linear mathematical function. The concept of increasing or decreasing linearity is used to describe the precedence relationship among a set of processes. The input for the model is a set of preemptible processes P, resident in a computer with a single shared memory and one or more processing elements. Each process P has a request time R, which is an arbitrary time at which P has been requested to be executed and a processing or computation time, C. For each P, a value function, V(t) is created where t is a time for which a value is to be determined and V defines the value to the system for completing P at time t. The nature of V is determined by which scheduling algorithm is used, BEValue1 or BEValue2.

These two algorithms take advantage of three value function and scheduling characteristics:

1. Given a set of processes (ignoring deadlines) with known values for completing them, it can be shown that a schedule in which the process with the highest value density V/C, (in which V is its value and C is its processing time as previously described) is processed first will produce a total value at every point in time at least as high as any other schedule. (i.e. a Value Density Schedule)
2. Given a set of processes with deadlines which can all be met (based on the sequence of the deadlines and the computation times of the processes), it can be shown that a schedule in which the process with the earliest deadline is scheduled first (i.e., an Earliest Deadline schedule) will always result in meeting all deadlines.
3. Most value functions of interest have their highest value occuring immediately prior to the critical time.

The BEValue1 Algorithm exclusively uses observation 1 above, and is therefore a simple greedy algorithm, scheduling first the process with the highest expected value density. It has been shown that this algorithm performs reasonably well in many cases in which the value function is a step function, or if the function is rapidly decreasing following the critical time, inspite of the fact it makes no use of critical time itself. The critical time does, of course, enter the algorithm through the expected value computation, which uses the value function and the assumed processing time distribution to compute an expected value. It was also shown by experimental results, that this algorithm fails most notably in step function situations where processor loads are low or at an average level, and a number of processes with close deadlines are in the request set.

The BEValue2 algorithm attempts to rectify this situation by the implementation of the following modification. This algorithm starts with a deadline-ordered sequence of available processes, which is then sequentially checked for its probability for overloading the processor. At any point in the sequence in which the overload probability passes a preset threshold, the process prior to the overload condition with the lowest value density, will be removed from the sequence. This process is repeated until the overload probability reaches an acceptable level. Because of this modification, this algorithm tends to out perform BEValue1 since it always meets deadlines as long as no processing overload occurs. However, when an overload condition occurs and gradually worsens, performance of this algorithm is similar to BEValue1. [Ref. 10: p.116]

7. DYNAMIC SCHEDULING OF TASK GROUPS

A more complex, yet extremely useful approach to process scheduling is described by [Ref. 15: pp. 166-174]. This research examined the problem of dynamic scheduling for groups of tasks in distributed real-time systems. The scheduling algorithm developed to meet this requirement is broken down into several smaller algorithms, a pre-processing algorithm, a distributed scheduling algorithm, and a compression algorithm.

The pre-processing algorithm divides processes into clusters and computes the required time to execute each cluster. Clusters are ordered into a precedence relationship based on these computations. This ordering is somewhat arbitrary and can be modified (through the the use of the compression algorithm) if necessary. Processes within a cluster are ordered according to real-time constraints by a method similar to that described by the earliest deadline approach. Based on this computation, this algorithm makes the decision whether or not there is enough processing time available to schedule a cluster of processes. If there is, a "dispatcher module" begins or enables the execution of the cluster.

Once a cluster begins executing, due to precedence constraints, processes within the cluster must synchronize in real-time in order to communicate with one another. When one process finishes executing, it sends an enabling message, as well as output data, to a successor process (the one which is next in the precedence ordering). A successor process can begin execution only after the enabling message from its predecessor has been received. Another module called the inter-task communication handler, is invoked each time a process finishes execution. This module evaluates incoming enabling messages and updates the number of finished predecessor processes when more than one is required for the execution of a particular successor task, and it sends enabling messages to successor tasks.

In the instance of a distributed system, the distributed scheduling algorithm is invoked when there is not enough processing time available to successfully execute a cluster. This algorithm attempts to find another location in the system for the cluster to be executed.

When it appears that a cluster cannot be successfully executed at any location, the compression algorithm is invoked. Because the computed execution time for a cluster is only an estimation, this algorithm is designed to compress the execution time for the entire cluster, or for individual processes when possible, within the cluster. [Ref. 15: pp. 167-169, 173].

8. A RECEIVER-INITITATED SCHEDULING STRATEGY

Another scheduling method is described in a comparison-oriented piece of research. Chang and Livny [Ref. 16: pp. 175-180] examined Sender-Initiated and Receiver-Initiated scheduling strategies in a multiprocessor environment. The Receiver-Initiated approach is of primary interest and works in the following way. Upon the completion of a job (process) the load of the processor is examined to determine if it is underloaded. When the number of jobs left in the queue is smaller than some preset threshold, the processor is tagged as underloaded. When this condition occurs, the underloaded processor polls other processors in the system to offer "help" (i.e. processing resources). This technique was proven to be an effective method for sharing and distributing resources among processors in a multi-processing environment. The basic idea appears to be a reasonable approach for sharing resources among processes as well.

9. APPLICATIONS OF THESE METHODS FOR THE CAPS SCHEDULERS

The foregoing scheduling methods were described to provide background information on the development of scheduling techniques and also to provide a basis for the development of the CAPS dynamic scheduler. Some of the techniques are also useful for describing the operation of the CAPS static scheduler and how the static and dynamic schedulers will interact in the execution support environment.

III. THE CAPS DYNAMIC SCHEDULER

A. SCHEDULING FUNCTIONS

Within the CAPS execution support system, the dynamic scheduler will perform several functions. First, it will act as the "run-time" executive that invokes, or starts the static scheduler and buffer preloading procedures for the translater. Second, it will create and invoke a schedule for the non-time critical operators of the PSDL prototype, third it will handle exceptions (both defined and undefined types) for all of the the CAPS components, and fourth it will handle both hardware and operator interrupts that may occur during prototype execution. These functions are illustrated by Figure 4 on page 24.

The proposed operation of the dynamic scheduler is outlined by the hierarchal description included as Appendix C. This design is based in part on Mok's "run-time scheduler" as described in [Ref. 12: pp. 5-17]. It provides the code for allocating resources in response to requests generated at run time, e.g. hardware or operator interrupts, and its scheduling decisions will not be dependent upon prior knowledge of future request times for processes to be executed. The specific functions it performs are described below.

1. THE RUN-TIME EXECUTIVE FUNCTION

At the start of prototype execution, the run-time executive function will invoke a procedure called PRELOADER for the translator. PRELOADER is a buffer initialization process required for implementation of PSDL data streams. The translator requires this process because buffers are regarded as "state machines" and must contain a certain value or be in a certain "state" at the start of prototype execution.

The static scheduler decomposes the prototype into a set of time critical and non-time critical operators. The result of this decomposition are files or "queues" of operators which are the input for the static schedule or the dynamic schedule. The run-time executive function will also invoke (start) the execution of the static schedule once it's created.

The schedule for time critical operators is based on the precedence relationships among the operators, and on the prototype's real-time constraints. The static scheduler creates a schedule that will ensure that both of these requirements are met. One of the

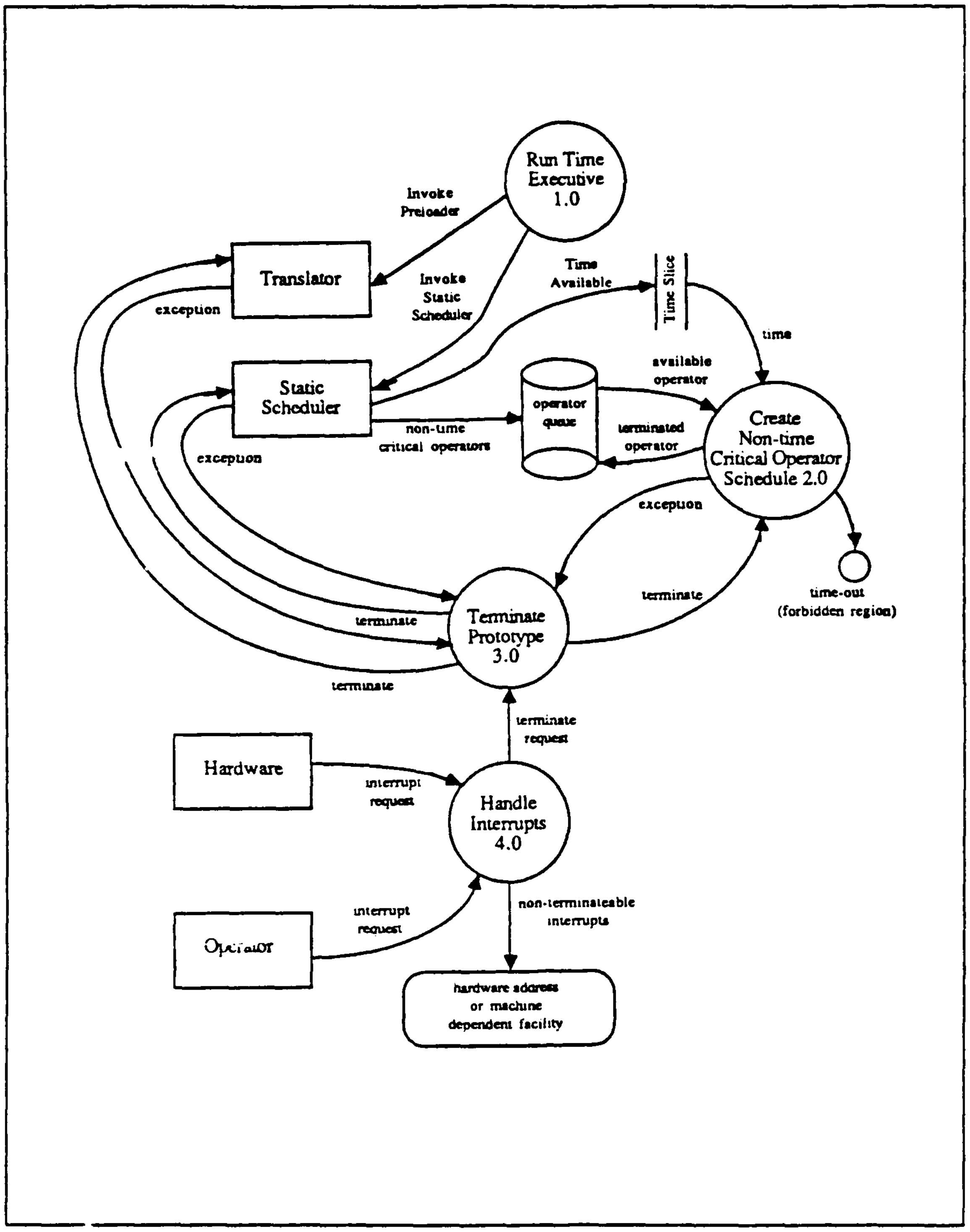


Figure 4. DYNAMIC SCHEDULER FUNCTIONS

scheduling approaches it uses to accomplish this is a blocking strategy similar to the method employed by Mok's Kernelized Monitor Model.

In the formulation of the static schedule, the static scheduler assumes worst case rather than average case processor utilization for meeting a given operator's processing requirements. The scheduling blocks will also contain periods of time between operators in which nothing is scheduled in order to ensure that precedence relationships are maintained (i.e. data flow requirements are met). These two conditions result in idle processing time that can be used by the dynamic scheduler to schedule and execute the prototype's non-time critical operators. The resulting spare processing capability will therefore occur unpredictably as shown by Figure 5 on page 26. It is then up to the dynamic scheduler to schedule non-time critical processes into these idle areas of the static schedule. This idea is similar to the "swapping" methodology employed by the "ED-PP" algorithm, and the "time-division multiplexing" approach within the rate monotonic algorithm.

2. THE CREATE NON-TIME CRITICAL OPERATOR SCHEDULE FUNCTION

When idle processing time is available for use by the dynamic scheduler, the steps illustrated by Figure 6 on page 27 will take place. The static scheduler will attempt to "rendezvous" with the dynamic scheduler in order to indicate or "send the message" that processing time (a "time slice") is available. This process is based on the "receiver-initiated" (poll-when-idle) strategy, and on the concepts of "inter-task communication" and "dispatcher" modules as described in the discussion of dynamic scheduling algorithms for distributed systems.

The dynamic scheduler must then determine (i.e. perform a compare operation) if there is enough time available in the time slice to execute a non-time critical process before the next scheduled start time of a time-critical scheduling block. This compare operation is analogous to an operation performed by the BEValue2 algorithm of the time-driven scheduling model. Recall that this algorithm makes a determination as to whether or not a given process will overload the processor. Similarly, the dynamic scheduler should determine whether or not a non-time critical process can be successfully executed within a given amount of time. If this is not possible, the process won't be scheduled. When there is enough time available, operators will be scheduled using one of the basic principles of the rate-monotonic algorithm. That is, an operator with the

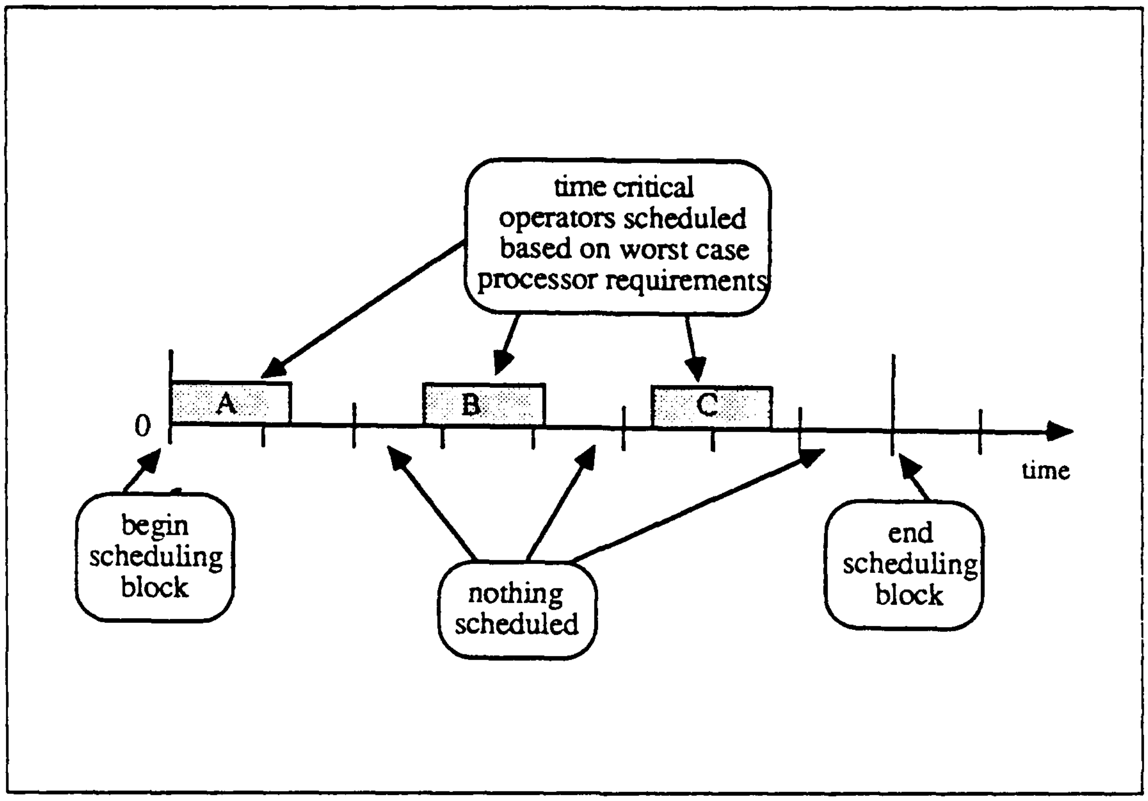

time critical operators scheduled based on worst case processor requirements
A
B
C
0
time
begin scheduling block
nothing scheduled
end scheduling block

Figure 5. STATIC SCHEDULER BLOCKING METHOD

shortest execution time will be considered to have the highest priority and will be scheduled for execution first.

This "priority" assignment is an arbitrary one since the processes to be executed are not time critical. The logic of this approach is simply to schedule as many non-time critical processes as possible into a "block" of idle time and it is based on the following assumptions:

1. Employment of a more complex scheme such as the creation of a "value density schedule", is unnecessary and would not effectively contribute to allotment of processing resources among the non-time critical processes.
2. Processes are independent of one another (i.e. there are no precedence relationships among the operators).
3. An execution time must be assigned to each of the operators during the specification phase of prototype development. The assigned execution time should not be confused with a "timing constraint", it is only meant to provide an estimate of the resources required for the execution of a non-time critical process.

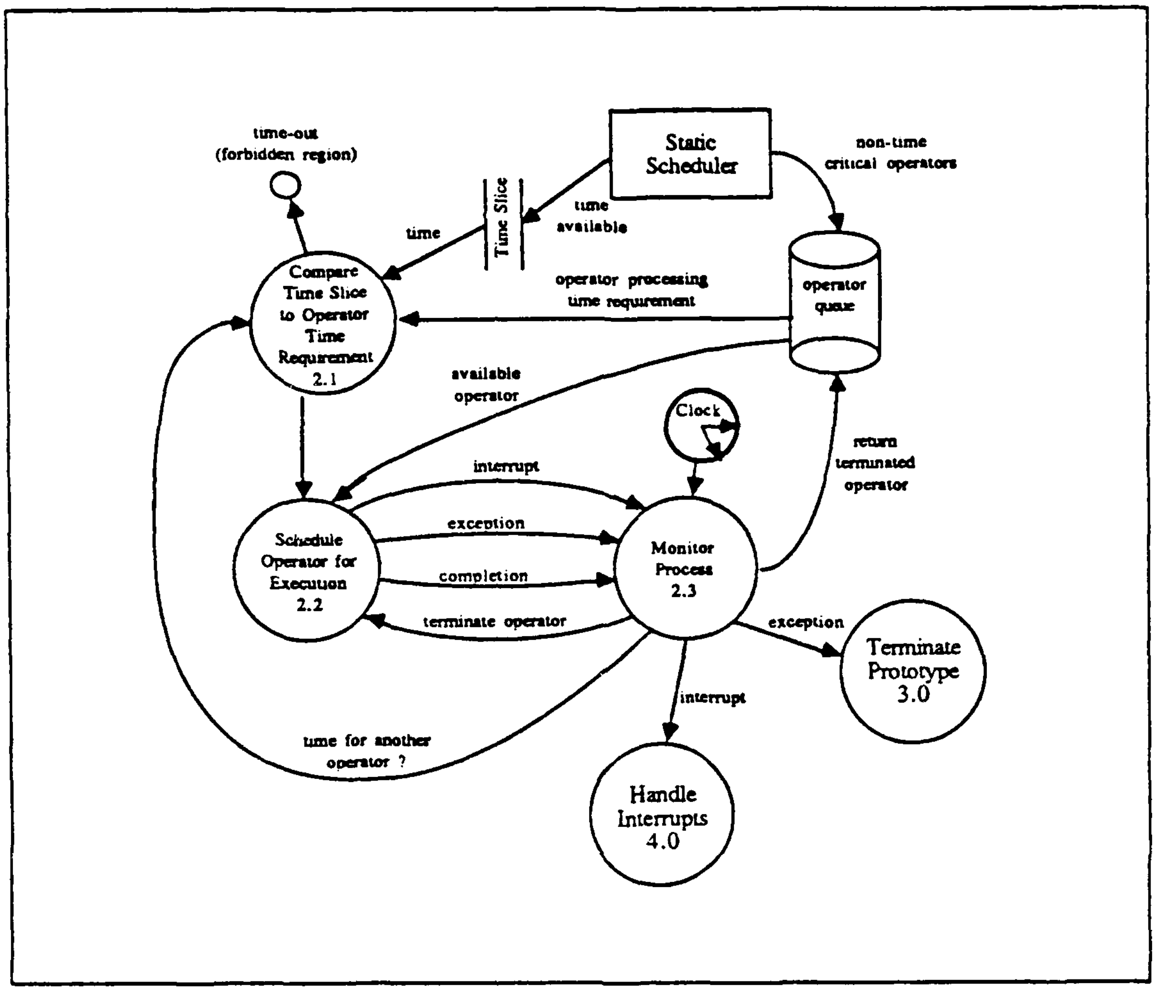


Figure 6. CREATION AND EXECUTION OF THE DYNAMIC SCHEDULE

4. Non-time critical processes will be sequenced in the "operator queue" based on a "shortest first" scheme. This sequencing will be performed by the static scheduler during the prototype decomposition operation.

For as long as time remains in an unused portion of the static schedule, the dynamic scheduler can schedule non-time critical processes for execution based on the preceding assumptions. When there is not enough time available to schedule the operator at the top of the queue (the operator with the shortest processing requirement), the dynamic scheduler will go into a "wait" state and allow the processor to remain idle until the start of the next static scheduling period. Allowing idle time in this instance is based on the idea of a "forbidden region" in the Kernelized Monitor Model. This forbidden region is necessary in order to ensure that a future deadline of the static schedule can be met. Allowing this idle time when using a rate monotonic approach also makes sense from a performance standpoint since utilization related research has indicated that processing efficiency tends to decline for processor loads above ln 2 (approximately 70%).

Even though a "compare" operation is performed to determine whether or not an operator can be completed within a given amount of time, the case may arise when a non-time critical process may exceed this amount of time. This cannot be allowed to occur since it would interfere with the static schedule and in effect, meeting the requirements of the system's real-time constraints. Therefore, execution of the non-time critical process must be preempted by some type of monitor.

The monitoring operation created to do this should track the status of an executing process relative to a system clock, and will terminate (preempt) a process in order for the next scheduling block within the static schedule to begin. When a process is terminated, it will be returned to the proper sequence position in the operator queue so that it can be rescheduled at another time. This monitoring process will also perform status monitoring with regard to completion of an operator i.e. it will "notify" the compare operation that the execution of a process is complete so that an attempt can be made to schedule another process. Finally, the monitor will call exception or interrupt handling procedures when the execution of a non-time critical process results in one of these two conditions.

3. THE TERMINATE PROTOTYPE FUNCTION

When exceptions occur as a result of processing performed by any of the three CAPS components, the terminate prototype function will be called. This function will perform the operations necessary to terminate the execution of the entire prototype. e.g. terminate whatever processes are executing at the time the exception occurs, and notify the CAPS user that an exception of a certain type has occured.

4. THE HANDLE INTERRUPTS FUNCTION

Two types of interrupts can occur while a prototype is executing, an operator interrupt and a hardware interrupt. Depending upon the nature of the interrupt, this function will call the terminate prototype function or it will initiate some other appropriate interrupt handling procedure. For example, in the instance of a hardware interrupt, instructions to go to a particular hardware address could be executed.

B. THE USE OF ADA

The dynamic scheduler will be implemented in Ada® as previously described. Appendix D provides a "skeleton" program based on the Ada® language in order to show some of the features of the language which are relevant for this implementation. For example, it demonstrates the use of an Ada® procedure. Recall that an Ada® procedure is a fundamental programming unit that encapsulates a series of statements.

This program also demonstrates the use of a task. A task in Ada® is based on the concept of communicating sequential processes. Tasks can be viewed as independent, concurrent operations that communicate with one another by passing "messages" [Ref. 5: pp. 68, 70]. This feature is particularly important to the CAPS execution support system as mentioned earlier because it provides the means for communication among each of the three CAPS components.

Another feature of the language included in this program is the instantiation of the generic package CALENDAR. CALENDAR has a predefined function, CLOCK that returns the time of day and exports a data type of time. This package provides a simple yet effective means for monitoring the execution time of an operator.

One other aspect of Ada® illustrated in Appendix D is an exception handling procedure. The Ada® language contains several predefined exceptions, and it also provides a user with the ability to define exceptions for a given application. For the CAPS, these user-defined exceptions will be be the predefined PSDL exceptions (e.g. FULL_BUFFER, EMPTY_BUFFER).

An exception is handled within the program unit where it is created (via a raise statement), or it can be sent (propagated) to another unit for handling. Since the dynamic scheduler is considered to be the run-time executive for the CAPS execution support system, it makes sense from an efficiency standpoint to handle exceptions at this "central" location within the execution environment.

The "centralization of control" logic also makes sense for the the handling of interrupts. Although not shown by the skeleton program, interrupt handling procedures can include an Ada® representation clause which allows the use of machine-dependent facilities. For example, an Ada® representation clause of the form "for FAIL use at 16#1FE#" as illustrated by [Ref. 5: p. 308] can be used. The hexadecimal number 16#1FE# represents some hardware or vector address.

One last language feature which should be mentioned, is a possible "file" structure for storing the non-time critical operators. Recall that this file (the "operator queue") is one of the results of the prototype decomposition performed by the static scheduler. Several different structures could be used depending upon which would provide the most effective means for performing input and output operations on processes during dynamic scheduling. One structure which is often used in Ada® to hold sorted data is a binary tree as illustrated by [Ref. 17: p. 150]. Other file structures which could be used include a linked list or a data stack. Implementation of any of these would allow the dynamic scheduler to perform the input/output operations required by its design.

IV. SUMMARY

A. THE QUESTIONS ANSWERED

This study attempted to meet two objectives:

1. Conceptual development of a dynamic scheduling component for the computer-aided design system CAPS
2. Interface of the dynamic scheduling component with the other two components of the CAPS execution support system

The focus on these objectives has resulted in the conceptual development of a four function dynamic scheduler. This design as outlined by Appendix C, demonstrates how the dynamic scheduler will interact with the translator and the static scheduler components within the CAPS execution support environment. Further, the scheduling approach proposed for the scheduling of a prototype's non-time critical provides a viable alternative for making effective use of idle processing resources that occur as a result of ensuring that a system's real-time constraints are met.

B. THE PROBLEMS THAT REMAIN

Future research for the CAPS dynamic scheduling problem needs to address several areas. An area of primary importance is a more detailed development of the conceptual design, including an examination of its feasibility given the assumptions its based on. Special attention should be placed on developing a more detailed description of the operations required for the "Create Non-time Critical Operator Schedule" function. Once this process is complete, the Ada® coding required to implement the dynamic scheduling functions can proceed.

Another area which needs to be addressed is the development of a "debugger" function for the dynamic scheduler as proposed by [Ref. 18] and [Ref. 2: p. 9]. The purpose of the debugger is to collect statistics on prototype behavior and to accept control of prototype execution when a PSDL exception occurs. (Recall that the initial dynamic design merely terminates prototype execution). The addition of this function would enhance, and at the same time, possibly reduce the number of iterative phases required during prototype development because of the additional control and information it provides to the designer.

The debugging function can be fairly conventional. For example, the ability to attach breakpoints to operators, which can be conditional with respect to a PSDL predicate (an "if" condition) could be included. Selected inputs or outputs of an operator should be traceable, resulting in a display of the values and their associated arrival or departure times. Commands for inserting and deleting values in data streams should also be provided.

The facilities for gathering statistics should include commands for monitoring both frequencies and timing information. Frequency statistics include the number of values that pass down a data stream, the number of times an exception occurs, etc. Timing statistics include minimum, average, standard deviation, and maximum times for the execution, response, or intervals between firings of an operator. These statistics are intended primarily for feasibility and performance studies. [Ref. 19: pp. 10 -13]

C. CAPS: AN EFFECTIVE DEVELOPMENT ALTERNATIVE

An example of an effort that would derive substantial benefit from the use of CAPS is the software development required for implementation of the Defense Switched Network (DSN). The implementation strategy that will be employed requires components and features to be adopted gradually, beginning with an initial capability based on today's voice network [Ref. 20: p. 11].

The DSN is the future Command and Control (C2) telecommunications network for the U.S. strategic armed forces. It is being designed to provide rapid, endurable, and economical telecommunications services to both high and low priority users. High priority users require immediate (i.e. real-time) service under the most difficult mission stress conditions. Low priority users require service for performing operational support activities such as logistics and personnel related functions which are not subject to the same type of real-time constraints. In order to meet these requirements, the network is planned to include more than 200 U.S. Government-owned communications switches in Europe and more than 60 U.S. Government-owned switches in the Pacific, as well as commercially leased switching and transmission services in the Western-Hemisphere and Hawaii. [Ref. 20: p. 6]

Comprehensive computer support that is highly reliable from both a security and a survivability standpoint, will be required to maintain control of this vast network. This computer support will assist in performing these network functions:

1. monitoring and surveillance to detect performance abnormalities automatically
2. implementing real-time controls that prevent switch or network congestion
3. analyzing traffic data to permit continuous optimal operation of the network

Computer aids that minimize personnel requirements will also be employed--locally and from remote locations--in administration, operations, maintenance, and network management of network elements. [Ref. 20: p. 6]

This diverse set of requirements illustrates why this development effort would be significantly enhanced by using CAPS, its prototyping methodology, and PSDL. This is especially true if the computer support systems are developed using Ada® as currently planned.

D. CONCLUSION

A primary advantage of CAPS for system development is that PSDL use for construction of an executable prototype would be much easier and simpler than direct use of Ada®. Additionally, executing a prototype (or prototypes) that demonstrates the functioning and interaction of modules within a complicated embedded system like the DSN, would significantly increase the confidence that the system can be built as planned. Using a prototype would also improve cost estimates since the cost of the intended system is generally proportional to the cost of a rapid prototype. [Ref. 19: p. 12]

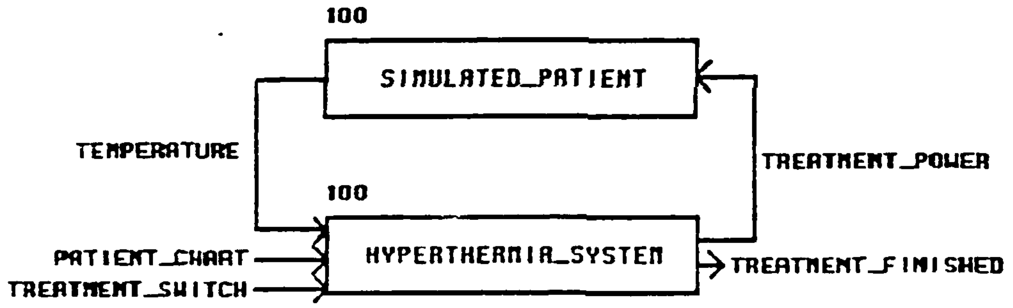
The conceptual development of the CAPS dynamic scheduler represents a significant step forward in meeting the demand for rapid development of reliable software for large real-time computer systems. Additionally, the proposed "shortest first" scheduling algorithm used by the dynamic scheduler could be effective for scheduling non-time critical processes in other real-time environments as well. This scheduling approach could prove to be an effective way for utilizing idle processing resources which are often wasted in large real-time systems.

APPENDIX A. A PSDL PROTOTYPE

This is an example of a PSDL prototype as it appears in [Ref. 4: pp. 27-40]. It was developed to model a simple system for treating brain tumors using hyperthermia.

```
OPERATOR brain_tumor_treatment_system
SPECIFICATION
 INPUT patient_chart:  medical_history,
    treatment_switch:  boolean
 OUTPUT treatment_finished:  boolean
 STATES temperature:  real
  INITIALLY 37.0
 DESCRIPTION
 {The brain tumor treatment system kills tumor cells
   by means of hyperthermia induced by microwaves.
 }
END

IMPLEMENTATION
 GRAPH
```



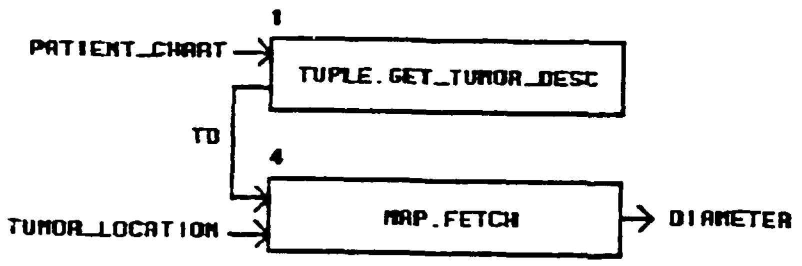
```
  DATA STREAM treatment_power:  real
  CONTROL CONSTRAINTS
   OPERATOR hyperthermia_system
    PERIOD 200 BY REQUIREMENTS shutdown
   OPERATOR simulated_patient
    PERIOD 200
  DESCRIPTION {paraphrased output}
 END


 TYPE medical_history
 SPECIFICATION
  OPERATOR get_tumor_diameter
   SPECIFICATION
    INPUTS patient_chart:  medical_history,
       tumor_location:  string
  OUTPUTS diameter:  real
  EXCEPTIONS no_tumor
  MAXIMUM EXECUTION TIME 5 ms
  DESCRIPTION
  {Returns the diameter of the tumor at a given location,
    produces an exception if no tumor at that location.
  }
  END


  KEYWORDS patient_charts, medical_records, treatment records,
     lab records
  DESCRIPTION
  {The medical history contains all of the disease and
    treatment information for one patient.  The operations
    for adding and retrieving information not needed by
    the hyperthermia system are not shown here.
  }
 END

 IMPLEMENTATION
  tuple {tumor_desc:  map-from:  string, to:  real{, ... }

  OPERATOR get_tumor_diameter
   IMPLEMENTATION
    GRAPH
```



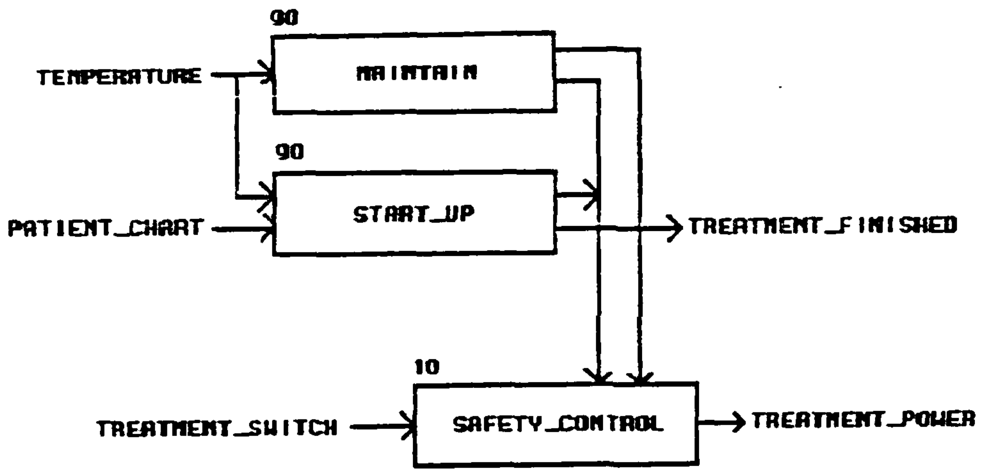
```
     DATA STREAM td:  tumor_descr
     CONTROL CONSTRAINTS
      OPERATOR map.fetch
       EXCEPTION no_tumor IF not(map.has(tumor_location, td))
    END


 END


 OPERATOR hyperthermia_system
 SPECIFICATION
  INPUT temperature:  real, patient_chart:  medical_history,
     treatment_switch:  boolean
  OUTPUT treatment_power:  real, treatment_finished:  boolean
  MAXIMUM EXECUTION TIME 100 ms
   BY REQUIREMENTS temperature_tolerance
```

```
MAXIMUM RESPONSE TIME 300 ms
 BY RFQUIREMENTS shutdown
KEYWORDS medical_equipment, temperature_control,
   hyperthermia, brain_tumors
DESCRIPTION
 {After the doctor turns on the treatment switch, the
   hyperthermia system reads the patient's medical record
   and turns on the microwave generator to heat the tumor
   in the patient's brain.  The system controls the power
   level to maintain the hyperthermia temperature of
   42.5 degrees C. for 45 minutes to kill the tumor cells.
   When the treatment is over, the system turns off the
   power and notifies the doctor.
 }
END

IMPLEMENTATION
 GRAPH
```



```
DATA STREAM estimated_power:  real
TIMER treatment_time
 CONTROL CONSTRAINTS
  OPERATOR start_up
   TRIGGERED IF temperature < 42.4
    BY REQUIREMENTS maximum_temperature
   STOP TIMER treatment_time
   RESET TIMER treatment_time IF temperature <= 37.0

  OPERATOR maintain
   TRIGGERED IF temperature >= 42.4
    BY REQUIREMENTS maximum_temperature
   START TIMER treatment_time
    BY REQUIREMENTS treatment_time, temperature_tolerance
   OUTPUT treatment_finished IF treatment_time >= 45 min
    BY REQUIREMENTS treatment_time
```

```
END


OPERATOR start_up
SPECIFICATION
 INPUT patient_chart:  medical_history, temperature:  real
 OUTPUT estimated_power:  real, treatment_finished:  boolean
  BY REQUIREMENTS startup_time
 MAXIMUM EXECUTION TIME 90 ms
  BY REQUIREMENTS temperature_tolerance
 DESCRIPTION
 {Extracts the tumor diameter from the medical history and
   uses it to calculate the maximum safe treatment power.
   Estimated power is zero if no tumor is present.  The
   treatment finished is true only if no tumor is present.
  }
END

IMPLEMENTATION Ada start_up
END


OPERATOR maintain
SPECIFICATION
 INPUT temperature:  real
 OUPUT estimated_power:  real, treatment_finished:  boolean
 MAXIMUM EXECUTION TIME 90 ms
  BY REQUIREMENTS temperature_tolerance
 DESCRIPTION
  { The power is controlled to keep the power between 42.4
   and 42.6 degrees C.
 }
END

IMPLEMENTATION Ada maintain
END


OPERATOR safety_control
SPECIFICATION
 INPUT treatment_switch, treatment_finished:  boolean
    estimated_power:  real
 OUTPUT treatment_power:  real
  BY REQUIREMENTS shutdown
 MAXIMUM EXECUTION TIME 10 ms
  BY REQUIREMENTS temperature_tolerance
 DESCRIPTION
 {The treatment power is equal to the estimated power
   if the treatment switch is true and treatment finished
   is false.  Otherwise the treatment power is zero.
 }
END

IMPLEMENTATION Ada start_up
END
```

APPENDIX B. PSDL GRAMMAR SUMMARY

This is a summary of PSDL grammar and language conventions as initially described in [Ref. 1: pp. 54-56] and further refined by [Ref. 6]. Several conventions are used for symbology in the grammar. [Square Braces] indicate optional items. { Curly Braces } indicate items which may appear zero or more times. Bold face type indicates a named terminal symbol which must appear in the program listing the programmer writes. "Double quotes" indicate character literals which must appear in the program listing. The "|" vertical bar indicates an exclusive-or selection. In this case the programmer selects one and only one of the items separated by the vertical bar.

As an example, the token timing_info is one of six mutually exclusive possibilities which may define the attribute token. The attribute token may appear zero or more times to define the interface token, which is a required attribute of the operator_spec token. Timing_info, if selected for attribute, may be empty, or it may contain one or more of the optional tokens allowed to define timing_info. Each of these tokens may appear no more than one time for a given instance of timing_info.

psdl = { component }

component = | data_type
| operator

data_type = **type** id type_spec type_impl

operator = **operator** id operator_spec operator_impl

type_spec = **specification** [type_decl] {op_spec_list} [functionality] **end**

op_spec_list = **operator** id operator_spec

operator_spec = **specification** interface [functionality] **end**

interface = {attribute [reqmts_trace]}

attribute = | generic_param
| input
| output
| states
| exceptions
| timing_info

generic_param = **generic** type_decl

input = **input** type_decl

output = **output** type_decl

states = **states** type_decl **initially** expression_list

exceptions = **exception** id_list

id_list = id { "," id }

timing_info = [**maximum execution time** time]
[**minimum calling period** time]
[**maximum response time** time]

time = **number** [unit]

unit = | **microsec** | **ms** | **sec** | **min** | **hours**

reqmts_trace = **by requirements** id_list

functionality = [keywords] [informal_desc] [formal_desc]

keywords = **keywords** id_list

informal_desc = **description** "{" text "}"

formal_desc = **axioms** "{" text "}"

type_impl = | **implementation Ada** id
| **implementation** type_name { op_impl_list } **end**

op_impl_list = **operator** id operator_impl

operator_impl = | **implementation Ada** id
| **implementation** psdl_impl

psdl_impl = data_flow_diagram
[streams]
[timers]
[control_constraints]
[informal_desc]
end

data_flow_diagram = **graph** { link }

link = id "." opid "->" id

opid = id [":" time]

streams = **data_stream** type_decl

type_decl = id_list ":" type_name { "," id_list ":" type_name }

type_name = | id
| id "[" type_decl "]"

timers = **timer** id_list

control_constraints = **control constraints** { constraint }

constraint = **operator** id

```
                    [triggered [trigger] [ "if" predicate] [reqmts_trace] ]
                    [period time [reqmts_trace] ]
                    [finish within time [reqmts_trace] ]
                    {output id_list if predicate [reqmts_trace] }
                    {exception id [if predicate] [reqmts_trace] }
                    {timer_op id [if predicate] [reqmts_trace] }

timer_op = | start | stop | read | reset

trigger =    | by all id_list
             | by some id_list

predicate =  | not predicate
             | predicate and predicate
             | predicate or predicate
             | expression_list
             | id ":" id_list

expression_list = expression { "," expression}

expression =        | number
                    | constant
                    | id
                    | type_name "." id "(" expression_list ")"
```

APPENDIX C. DYNAMIC SCHEDULER FUNCTIONS

1.0 Run-Time Executive

- 1.1 Invoke Translator Preloader Procedure
- 1.2 Invoke Static Scheduler

2.0 Create Non-Time Critical Operator Schedule

- 2.1 Compare Time Slice to Operator Queue Time Requirement
  - 2.1.1 Find top of queue (operator with shortest time requirement)
  - 2.1.2 Subtract operator time requirement from time slice
  - 2.1.3 When result of subtraction > 0, send time available message to execute operator function
  - 2.1.4 When result of subtraction < 0, let processor idle until start of next static schedule requirement
- 2.2 Schedule Operator
  - 2.2.1 Schedule available operator from operator queue for execution
  - 2.2.2 Send completion message to monitor
  - 2.2.3 Send exception message to monitor
  - 2.2.4 Send hardware/interrupt message to monitor
- 2.3 Monitor Process
  - 2.3.1 Monitor execution time of operators
  - 2.3.2 Terminate operator if available processing time is exceeded
  - 2.3.3 When operator completes execution, send message to compare operation to see if more execution time is available
  - 2.3.4 When exception occurs during dynamic schedule processing, call terminate prototype function
  - 2.3.5 When interrupt occurs during dynamic schedule processing, call handle interrupts function

3.0 Terminate Prototype

- 3.1 Terminate Translator
- 3.2 Terminate Static Scheduler
- 3.3 Terminate Dynamic Scheduler

4.0 Handle Interrupts

- 4.1 Send terminate request to terminate prototype
- 4.2 Send non-terminatable request to appropriate location

APPENDIX D. PSEUDO-CODE FOR AN ADA PROGRAM

This pseudo-code illustrates some useful features of the Ada® programming language (Ada® is a registered trademark of the United States Government, Ada Joint Programming Office). A detailed description of how these features can be implemented in an Ada® program appears in [Ref. 5]

```
--Two hyphens indicate the start of a comment in the Ada language.
--Four hyphens within this pseudo-code are used to enhance
--readability and to indicate the absence of formal
--parameters, statements, or other features of the language
--that are required by an actual program

    ----
    ----

        with OPERATOR_QUEUE;  --the operator queue of
                              --non-time critical processes
                              --will be created by the
                              --static scheduler

    ----
    ----

        with CALENDAR;  --the Ada language definition
                        --includes the package CALENDAR
                        --with a predefined function,
                        --CLOCK that returns the time
                        --of day and exports a data type
                        --of time

    ----
    ----

procedure DYNAMIC_SCHEDULER is

    ----
    ----

      declare

    ----
    ----

      FULL_BUFFER:       exception;  --when an exception is
      EMPTY_BUFFER:      exception;  --raised within an Ada
      OVER_TIME:         exception;  --program unit, it is
      PSDL_EXCEPTION:    exception;  --propogated to a level
```

```
                                        --where it can be handled

  ----
  ----

  type READY      is text;        --the text types indicate the
  type SCHEDULE   is text;        --different messages exchanged
  type TIME_SLICE is text;        --during a rendezvous

  ----
  ----

  PRELOAD    :  READY;
  SCHEDULE   :  CREATED;
  IDLE       :  TIME_SLICE;

  ----
  ----

  procedure PRELOADER;   --PRELOADER will be some actions
                         --that will invoke buffer
                         --initialization procedures for the
                         --translator


  procedure START;   --START will consist of some actions
                     --to start the execution of the
                     --static schedule

  ----
  ----


 procedure CREATE_SCHEDULE is   --the procedure that will
                                --create a schedule for
                                --the non-time critical
                                --operators

  ----
  ----

    use CALENDAR;
    use OPERATOR_QUEUE;

  ----
  ----

         TIME, OPERATOR_TIME_REQUIREMENT  :  TIME_SLICE;

  ----
  ----

  begin


       --COMPARE_OPERATION
 ----
 ----

          --find top of OPERATOR_QUEUE (operator with shortest
          --time requirement)

          --select this operator and compare its execution
          --time with TIME_SLICE in order to determine
          --if enough time is available to
          --execute this non-time critical process

          --while enough time is available, in a given
          --TIME_SLICE, schedule processes for execution

          --else let the processor idle till start
          --of next static scheduling block

 ----
 ----

      --MONITOR_PROCESS  --implement a process to monitor
                         --status of executing non-time
                         --critical operators (time, completion, etc.)
                         --using the generic package CALENDAR

 ----
 ----

end CREATE_SCHEDULE;

 ----
 ----

 task RUN_TIME_EXECUTIVE is  --an Ada task is an effective
                             --method for implementing the
                             --the run-time executive function
                             --because it provides a means for
                             --communication among the three
                             --execution support system commponents

                             --entry and accept provide the
                             --means for "two way"
                             --communications among the three
                             --execution support system components

    entry TRANSLATOR (PRELOAD  : in READY);

                             --the communications path from
                             --the dynamic scheduler
                             --to the translator which will be
                             --used to invoke the buffer
                             --preloader procedure

    entry STATIC_SCHEDULER (SCHEDULE  : in CREATED);
```

```
                                   --the communications path between
                                   --the dynamic scheduler and the
                                   --static scheduler which will be
                                   --used to invoke (start) the
                                   --execution of the static schedule


   entry IDLE_TIME (IDLE  : in  TIME_SLICE);

                                   --the communications path between
                                   --the dynamic scheduler and the
                                   --static scheduler which will be
                                   --used to indicate to the dynamic
                                   --scheduler when idle time is
                                   --available
end;

----
----


task body RUN_TIME_EXECUTIVE is


   begin

----
----

   accept TRANSLATOR (PRELOAD  : in READY) do PRELOADER;

                               --PRELOADER will be some actions that
                               --will invoke buffer initialization
                               --instructions

----
----

   accept STATIC_SCHEDULER (SCHEDULE  : in CREATED) do START;

                               --START will consist of some actions
                               --to start the execution of the static
                               --schedule

----
----

   accept IDLE_TIME (IDLE  : in  TIME_SLICE) do CREATE_SCHEDULE;

                               --when idle time is available, the
                               --dynamic scheduler can schedule
                               --non-time critical processes for
                               --execution during a given
                               --"time slice"
```

```
----
----

   end RUN_TIME_EXECUTIVE;

----
----

begin

----
----

    RUN_TIME_EXECUTIVE.TRANSLATOR (PRELOADER);

    RUN_TIME_EXECUTIVE.STATIC_SCHEDULER (START);

    RUN_TIME_EXECUTIVE.IDLE_TIME (CREATE_SCHEDULE);

----
----

  --when an exception occurs, the generic procedure TEXT_IO
  --and an application specific procedure such as PUT_LINE
  --can be used indicate to the CAPS user what the nature
  --of the exception is

----
----

  exception
        when FULL_BUFFER=>

             TEXT_IO.PUT_LINE ("An attempt was made to
                               update a full buffer");
             TERMINATE_PROTOTYPE;

             --using the Ada generic package TEXT_IO,
             --and a user written procedure PUT_LINE,
             --a message as shown will appear on the
             --user's screen and prototype execution
             --will be terminated when an exception is
             --raised.
        end;

----
----

  exception
        when EMPTY_BUFFER=>

             TEXT_IO.PUT_LINE ("An attempt was made to
                               read data from an empty
                               buffer");
             TERMINATE_PROTOTYPE;
```

```
            end;

    ----
    ----

      exception
            when OVER_TIME=>

                 TEXT_IO.PUT_LINE ("A PSDL operator has
                                   exceeded maximum
                                   execution time");
                 TERMINATE_PROTOTYPE;


             end;

    ----
    ----

      exception
            when  PSDL_EXCEPTION=>

                  TEXT_IO.PUT_LINE ("An undefined PSDL
                                    exception
                                    has occurred");
                  TERMINATE_PROTOTYPE;

             end;

    ----
    ----

end DYNAMIC_SCHEDULER;

    ----
    ----
```

LIST OF REFERENCES

1. Luqi, *Rapid Prototyping for Large Software System Design,* Ph.d Thesis, University of Minnesota, Duluth, MN, May 1986.

2. Luqi, and Ketabchi, M., *A Computer Aided Prototyping System,* Technical Report NPS52-87-011, Naval Postgraduate School, Monterey, CA, April 1987.

3. Luqi, and Ketabchi, M., "A Computer Aided Prototyping System", *IEEE Software,* IEEE Computer Society Press, Washington, D.C., 66-72, March 1988.

4. Luqi, Berzins, V., Yeh, R., *A Prototyping Language for Real-Time Software,* Technical Report NPS52-87-010, Naval Postgraduate School, Monterey, CA, April 1987.

5. Booch, G., *Software Engineering with Ada,* The Benjamin/Cummings Publishing Company, Inc., Menlo Park, CA, 1987.

6. Moffitt, C. R., *A Language Translator For A Computer Aided Rapid Prototyping System,* M.S. Thesis, Naval Postgraduate School, Monterey, CA, March 1988.

7. O'Hern, J. T., *A Conceptual Level Design For A Static Scheduler For Hard Real-Time Systems,* M.S. Thesis, Naval Postgraduate School, Monterey, CA, March 1988.

8. Janson, D. M., *A Static Scheduler For The Computer Aided Prototyping System: An Implementation Guide,* M.S. Thesis, Naval Postgraduate School, Monterey, CA, March 1988.

9. Davari, S. and Dhall, S. K., "An On Line Algorithm for Real-Time Tasks Allocation", *IEEE Real-Time Systems: Proceedings of the Symposium in New Orleans, Lousiana, December 2-4, 1986,* IEEE Computer Society Press. Washington, D.C., 194-199, 1987.

10. Jensen, E. D., Locke, C. D., Tokuda, H., "A Time-Driven Scheduling Model for Real-Time Operating Systems", *IEEE Real-Time Systems: Proceedings of the Symposium in San Diego, California, December 3-6, 1985,* IEEE Computer Society Press. Washington, D.C., 112-122, 1986.

11. Mok, A. K., "The Decomposition of Real-Time System Requirements into Process Models", *IEEE Real-Time Systems: Proceedings of the Symposium in Austin, Texas, December 4-6, 1984,* IEEE Computer Society Press. Washington, D.C., 125-133, 1985.

12. Mok, A. K., "The Design of Real-Time Programming Systems Based on Process Models", *IEEE Real-Time Systems: Proceedings of the Symposium in Austin, Texas, December 4-6, 1984,* IEEE Computer Society Press. Washington, D.C., 5-17, 1985.

13. Mok, A. K., and Sutanthavibul, S., "Modeling and Scheduling of Dataflow Real-Time Systems", *IEEE Real-Time Systems: Proceedings of the Symposium in San Diego, California, December 3-6, 1985,* IEEE Computer Society Press. Washington, D.C., 178-187, 1986.

14. Sha, L., Lehoczky, J. P., Rajkumar, R., "Solutions for Some Practical Problems in Prioritized Preemptive Scheduling", *IEEE Real-Time Systems: Proceedings of the Symposium in New Orleans, Lousiana, December 2-4, 1986,* IEEE Computer Society Press. Washington, D.C., 181-191, 1987.

15. Cheng, S., Stankovic, J. A., Ramamritham, K., Dynamic "Scheduling of Groups of Tasks with Precedence Constraints in Distributed Hard Real-Time Systems", *IEEE Real-Time Systems: Proceedings of the Symposium in New Orleans, Louisiana, December 2-4, 1985,* IEEE Computer Society Press. Washington, D.C., 166-174, 1987.

16. Chang, H., and Livny, M., "Distributed Scheduling under Deadline Constraints: Comparison of Sender-initiated and Receiver-initiated Approaches", *IEEE Real-Time Systems: Proceedings of the Symposium in New Orleans, Louisiana, December 2-4, 1986,* IEEE Computer Society Press. Washington, D.C., 175-180, 1987.

17. Bray, G., and Pokrass, D., *Understanding Ada--A Software Engineering Approach,* John Wiley and Sons, Inc., New York, NY, 1985.

18. Luqi, "Execution of Real-Time Prototypes", *ACM First International Workshop on Computer Aided Software Engineering*, Cambridge, MA, 870-884, May 1987.

19. Luqi, *Execution of Real-Time Prototypes,* Technical Report NPS52-87-012, Naval Postgraduate School, Monterey, CA, April 1987.

20. Defense Communications Agency, *Defense Switched Network,* The Defense Communications Agency, Washington, D.C., 1987.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, CA 93943-5002	2
3.	Office of the Chief of Naval Operations Code OP-941 Washington, DC 20350	1
4.	Commander, Naval Telecommunications Command Naval Telecommunications Command Headquarters 4401 Massachusetts Avenue N. W. Washington, DC 20350	1
5.	Naval Telecommunications System Integration Center Naval Communications Unit Washington Washington, DC 20397-5340	1
6.	Ada Joint Program Office OUSDRE(R&AT) The Pentagon Washington, DC 20301	1
7.	Commander, Naval Data Automation Command Washington Navy Yard Washington, D.C. 20374-1662	1
8.	Chief of Naval Research Office of the Chief of Naval Research Atten. CDR. Michael Gehl Code 1224 Arlington, VA 22217-5000	1
9.	Professor LUQI, Code 52LQ Naval Postgraduate School Monterey, CA 93943	1
10.	LCDR Barry A. Frew, USN, Code 54FW Naval Postgraduate School Monterey, CA 93943	1
11.	Defense Communications Agency Attn: LT Susan L. Eaton, Code B531 Washington, DC 20305	3

12. LT Charlie R. Moffitt, USN 1
Department Head Class #104
SWOSCOLOM, Bldg. 446
Newport, RI 02841-5012

13. Office of the Chief of Naval Operations 1
Code OP-945
Washington, DC 20350

14. Professor D. C. Boger, Code 54BO 1
Naval Postgraduate School
Monterey, CA 93943